Chapter 14

Structures, the Multivariable

In This Chapter

- Creating structures
- Declaring structure variables
- Assigning values to a structure variable
- Building structure arrays
- Putting one structure inside another

Individual variables are perfect for storing single values. When you need more of one type of a variable, you declare an array. For data that consists of several different types of variables, you mold those variable types into something called a structure. It's the C language's method of creating a variable buffet.

Hello, Structure

I prefer to think of the C language structure as a *multivariable,* or several variables rolled into one. You use structures to store or access complex information. That way, you can keep various `int`, `char`, `float` variables, and even arrays, all in one neat package.

Introducing the multivariable

Some things just belong together. Like your name and address or your bank account number and all the money that's supposedly there. You can craft such a relationship in C by using parallel arrays or specifically named variables. But that's clunky. A better solution is to employ a structure, as demonstrated in Listing 14-1.

Listing 14-1: One Variable, Many Parts

```
#include <stdio.h>

int main()
{
    struct player
    {
        char name[32];
        int highscore;
    };
    struct player xbox;

    printf("Enter the player's name: ");
    scanf("%s",xbox.name);
    printf("Enter their high score: ");
    scanf("%d",&xbox.highscore);

    printf("Player %s has a high score of %d\n",
            xbox.name,xbox.highscore);
    return(0);
}
```

Exercise 14-1: Without even knowing what the heck is going on, type Listing 14-1 into your editor to create a new program. Build and run.

Here's how the code in Listing 14-1 works:

Lines 5 through 9 declare the `player` structure. This structure has two members — a `char` array (string) and `int` — declared just like any other variables, in Lines 7 and 8.

Line 10 declares a new variable for the `player` structure, `xbox`.

Line 13 uses `sacnf()` to fill the `name` member for the `xbox` structure variable with a string value.

Line 15 uses `scanf()` to assign a value to the `highscore` member in the `xbox` structure.

The structure's member values are displayed at Line 17 by using a `printf()` function. The function is split between two lines with a backslash at the end of Line 17; variables for `printf()` are set on Line 18.

Understanding struct

A structure isn't a variable type. Instead, think of it as a frame that holds multiple variable types. In many ways, a structure is similar to a record in a database. For example:

```
Name
Age
Gambling debt
```

These three items can be fields in a database record, but they can also be members in a structure: `Name` would be a string; `Age`, an integer; and `Gambling Debt`, an unsigned floating-point value. Here's how such a record would look as a structure in C:

```
struct record
{
    char name[32];
    int age;
    float debt;
};
```

`struct` is a C language keyword that introduces a structure. Or you can look at it as defining or creating a new structure.

`record` is the name of the new structure being created. It's not a variable — it's a structure type.

Within the curly brackets dwell the structure's members, the variables contained in the named structure. The `record` structure type contains three member variables: a string `name`, an `int` named `age`, and a `float` value, `debt`.

To use the structure, you must declare a structure variable of the structure type you created. For instance:

```
struct record human;
```

This line declares a new variable of the `record` structure type. The new variable is named *`human`*.

Structure variables can also be declared when you define the structure itself. For example:

```
struct record
{
    char name[32];
    int age;
    float debt;
} human;
```

These statements define the `record` structure *and* declare a `record` structure variable, *`human`*. Multiple variables of that structure type can also be created:

```
struct record
{
    char name[32];
    int age;
    float debt;
} bill, mary, dan, susie;
```

Four `record` structure variables are created in this example. Every variable has access to the three members defined in the structure.

To access members in a structure variable, you use a period, which is the *member operator*. It connects the structure variable name with a member name. For example:

```
printf("Victim: %s\n",bill.name);
```

This statement references the `name` member in the *`bill`* structure variable. A `char` array, it can be used in your code like any other `char` array. Other members in the structure variable can be used like their individual counterparts as well:

```
dan.age = 32;
```

In this example, the `age` member in the structure variable *`dan`* is set to the value `32`.

Exercise 14-2: Modify the source code from Listing 14-1 so that another member is added to the player structure, a `float` value indicating `hours` played. Fix up the rest of the code so that the new value is input and displayed.

Filling a structure

As with other variables, you can assign values to a structure variable when it's created. You must first define the structure type and then declare a structure variable with its member values preset. Ensure that the preset values match the order and type of members defined in the structure, as shown in Listing 14-2.

Listing 14-2: Declaring an Initialized Structure

```
#include <stdio.h>

int main()
{
    struct president
    {
        char name[40];
        int year;
    };
    struct president first = {
        "George Washington",
        1789
    };

    printf("The first president was %s\n",first.name);
    printf("He was inaugurated in %d\n",first.year);

    return(0);
}
```

Exercise 14-3: Create a new program by typing the source code from Listing 14-2 into the editor. Build and run.

You can also declare a structure and initialize it in one statement:

```
struct president
{
    char name[40];
    int year;
} first = {
    "George Washington",
    1789
};
```

Exercise 14-4: Modify your source code from Exercise 14-3 so that the structure and variable are declared and initialized as one statement.

Though you can declare a structure and initialize a structure variable as just shown, you can get away with that trick only once. You cannot use the technique to declare the second structure variable, which must be done the traditional way, as shown in Listing 14-2.

Exercise 14-5: Add another `president` structure variable, `second`, to your code, initializing that structure with information about the second president, John Adams, who was inaugurated in 1797. Display the contents of both structures.

Making an array of structures

Creating individual structure variables, one after the other, is as boring and wasteful as creating a series of any individual variable type. The solution for multiple structures is the same as for multiple individual variables: an array.

A structure array is declared like this:

```
struct scores player[4];
```

This statement declares an array of `scores` structures. The array is named `player`, and it contains four structure variables as its elements.

The structures in the array are accessed by using a combination of array and structure notation. For example:

```
player[2].name
```

The variable in the preceding line accesses the `name` member in the third element in the `player` structure array. Yes, that's the third element because the first element would be referenced like this:

```
player[0].name
```

Arrays start numbering with the element `0`, not element `1`.

Line 10 in Listing 14-3 declares an array of four `scores` structures. The array is named `player`. Lines 13 through 19 fill each structure in the array. Lines 21 through 27 display each structure's member values.

Listing 14-3: Arrays of Structures

```
#include <stdio.h>

int main()
{
    struct scores
    {
        char name[32];
        int score;
    };
    struct scores player[4];
    int x;

    for(x=0;x<4;x++)
    {
        printf("Enter player %d: ",x+1);
        scanf("%s",player[x].name);
        printf("Enter their score: ");
        scanf("%d",&player[x].score);
    }

    puts("Player Info");
    printf("#\tName\tScore\n");
    for(x=0;x<4;x++)
    {
        printf("%d\t%s\t%5d\n",
            x+1,player[x].name,player[x].score);
    }
    return(0);
}
```

Exercise 14-6: Type the source code from Listing 14-3 into your editor. Build and run the program. Try to keep the scores to fewer than five digits so that they line up properly.

Exercise 14-7: Add code to Listing 14-3 so that the display of structures is sorted with the highest score listed first. Yes, you can do this. Sorting an array of structures works just like sorting any other array. Review Chapter 12 if you suddenly lose your nerve.

Here's a hint, just because I'm a nice guy. Line 27 of my solution looks like this:

```
player[a]=player[b];
```

You can swap structure array elements just as you can swap any array elements. You don't need to swap the structure variable's members.

Weird Structure Concepts

I'll admit that structures are perhaps the weirdest type of variable in the C language. The two steps required to create them are unusual, but the dot method of referencing a structure's member always seems to throw off beginning programmers. If you think that, beyond those two issues, structures couldn't get any odder, you're sorely mistaken.

Putting structures within structures

It's true that a structure holds C language variables. It's also true that a structure is a C language variable. Therefore, it follows that a structure can hold another structure as a member. Don't let this type of odd thinking confuse you. Instead, witness the example shown in Listing 14-4.

Listing 14-4: A Nested Structure

```
#include <stdio.h>
#include <string.h>

int main()
{
    struct date
    {
        int month;
        int day;
        int year;
    };
    struct human
    {
        char name[45];
        struct date birthday;
    };
    struct human president;

    strcpy(president.name,"George Washington");
    president.birthday.month = 2;
    president.birthday.day = 22;
    president.birthday.year = 1732;

    printf("%s was born on %d/%d/%d\n",
            president.name,
            president.birthday.month,
            president.birthday.day,
            president.birthday.year);

    return(0);
}
```

Listing 14-4 declares two structure types: `date` at Line 6 and `human` at Line 12. Within the `human` structure's declaration, at Line 15 you see the `date` structure variable *`birthday`* declared. That's effectively how one structure is born inside another.

Line 17 creates a `human` structure variable, *`president`*. The rest of the code fills that structure's members with data. The method for accessing a nested structure's members is shown in Lines 20 through 22.

The structure's variable names are used; not the name that's used to declare the structure.

Exercise 14-8: Type the source code from Listing 14-4 into your editor. Build and run the program.

Exercise 14-9: Replace the *`name`* member in the `human` structure with a nested structure. Name that structure `id` and have it contain two members, `char` arrays, `first` and `last`, for storing an individual's first and last names. If you do everything correctly, the reference to the president's name will be the variables *`president.name.first`* and *`president.name.last`*. Be sure to assign values to these variables in your code and display the results.

Passing a structure to a function

As a type of variable, it's entirely possible for a function to eat a structure and cough it up. However, this situation requires that the structure be declared as a global variable. That's because if you declare a structure within a function, and `main()` is a function, the definition is available only in that function. Therefore, the declaration must be made globally so that it's available to all functions in the code.

The topic of global variables is introduced in Chapter 16. It's not that complex, but suffice it to say that I put off until then discussing how to pass and return structures to and from functions.